# Integrating Functional and Imperative Parallel Programming: CC++ Solutions to the Salishan Problems*

John Thornley
Computer Science Department
California Institute of Technology
Pasadena, California 91125, USA
john-t@cs.caltech.edu

December 6, 1993

**Abstract**

*We investigate the practical integration of functional and imperative parallel programming in the context of a popular sequential object-based language. As the basis of our investigation, we develop solutions to the Salishan Problems, a set of problems intended as a standard by which to compare parallel programming notations. The language that we use is CC++, C++ extended with single-assignment variables, parallel composition, and atomic functions. We demonstrate how deterministic parallel programs can be written that are identical—except for the addition of a few keywords—to sequential programs that satisfy the same specifications.*

## 1 Introduction

The difficulty of developing reliable, maintainable, and portable parallel programs is the major obstacle to the more widespread application of parallel programming. To overcome this obstacle, our general goal is to devise high level programming notations that allow parallel programs to be written close to the level of problem specification, yet still be compiled to

*This work was supported in part by Air Force Office of Scientific Research grant AFOSR-91-0070.

execute efficiently across a variety of different single and multiple processor architectures. The specific contribution of the work that we present in this report is to show how functional and imperative parallel programming can be integrated in a simple and consistent manner within the framework of a popular sequential language that supports object-based data abstraction. We are motivated by the observation that the strengths of functional and imperative programming are complementary with respect to our goal.

Imperative parallel programming [1] expresses a computation operationally, as a group of cooperating concurrent processes, each with its own state and sequential thread of control. Processes communicate and synchronize using, for example, message passing, shared memory with locks, or remote procedure calls. The strength of imperative parallel programming is in representing explicitly concurrent entities and operations from the problem specification, e.g., modeling or controlling physical systems, computer operating systems, and human interfaces. Its weakness is the complexity of expressing parallelism that is not explicit in the problem specification, solely for execution performance on multiprocessor architectures.

Functional parallel programming [2, 3] expresses a computation nonoperationally, as a deterministic mapping from input values onto output values. Programs can be executed in any manner such that operands are evaluated before they are needed. Exploitation of parallelism is mostly the responsibility of the compiler and runtime system, not the programmer. Recent work demonstrates that functional programming can provide execution performance comparable to that of imperative programming [4]. The strength of functional parallel programming is in representing parallel algorithms for problem specifications that are given as a functional mapping from inputs onto outputs, e.g., many scientific problems. Its weakness is the difficulty of representing state, sequence, and concurrency that are explicit in the specification of many problems.

This report investigates the practical integration of the complementary strengths of functional and imperative parallel programming. Our work differs from other work that integrates functional and imperative parallel programming [5, 6] in that we choose to build upon an established sequential object-based language. The language that we use is CC++ [7, 8], a simple extension of C++ [9] that supports both functional and imperative parallel programming. Previous work with CC++ has concentrated on the definition and use of imperative parallel programming libraries [10]. To evaluate the benefits of the integrated use of both functional and imperative parallel programming in this context, we develop solutions to the Salishan

Problems [11], a representative set of problems intended as a standard by which to compare parallel programming notations.

The remainder of this report is organized as follows: in Section 2 we briefly describe CC++; in Section 3 we introduce the Salishan Problems; in Sections 4 through 7 we present solutions to the individual problems; and in Section 8 we conclude with an evaluation of the results of our investigation. In Appendixes A through D we give the complete text of our solution programs.

## 2 The CC++ Language

CC++ (Compositional C++) is C++ with six extensions. Of those extensions, we use the following four in this report:

1. **Sync Types:**

   A variable of a **sync** type initially has a special undefined value, and can be assigned a value at most once. Evaluation of an undefined **sync** variable suspends until the variable is assigned a value. Types that are not **sync** are referred to as "mutable" types.

   Examples:

   ```
   sync int i;         // sync int
   sync int a[N];      // array of sync int
   sync int* p;        // mutable pointer to sync int
   int *sync q;        // sync pointer to mutable int
   sync int *sync r;   // sync pointer to sync int
   ```

2. **Parallel Blocks:**

   The statements in a **par**-block are executed as parallel processes. Execution of a **par**-block terminates when execution of all its statements has terminated.

   Example:

   ```
   sync int a, b, c;

   par {
       c = a + b;
       b = 2;
       a = 1;
   }
   // Assert: a == 1 and b == 2 and c == 3.
   ```

3. **Parallel For Statements:**

   The iterations of a **parfor** statement are executed as parallel processes. Execution of a **parfor** statement terminates when execution of all its iterations has terminated.

   Example:

```
sync int a[N];

par {
    parfor (int i = 0; i < N - 1; i++)
        a[i] = a[i + 1] - 1;
    a[N - 1] = N - 1;
}
// Assert: ∀ i : 0 ≤ i < N : a[i] == i.
```

4. **Atomic Functions:**

   The execution of an **atomic** function is not interleaved with the execution of any other **atomic** function of the same object.

   Example:

```
int count[100];

atomic void increment(int index)
{
    count[index] = count[index] + 1;
}
```

The two extensions that we do not use in this report are constructs for the unstructured spawning of processes and for the distribution of computations across multiple address spaces. A complete definition of the syntax and semantics of CC++ is given by [8].

# 3 The Salishan Problems

The Salishan Problems are a set of four problems proposed as a standard by which to compare parallel programming notations. The problem set was originally defined at the 1988 Salishan High-Speed Computing Conference. At that conference, invited speakers presented solutions to the problems in eight different parallel programming languages. Loosely categorized, those languages were: Ada and Occam (imperative); Haskell, Id, and Sisal (functional); C* (data-parallel); PCN and Scheme (combined imperative and functional). The Salishan Problems and the original eight sets of solutions are published in [11].

Three of the four problems—Hamming's Problem, the Paraffins Problem, and the Skyline Matrix Problem—are functional mappings from input values onto output values, without any concurrency in their problem specifications. For these problems we write deterministic functional parallel programs using **sync** variables, **par**-blocks, and **parfor** statements. The remaining problem—the Doctor's Office Problem—is a model of the asynchronous interactions within a system of concurrent entities. For this problem we write a nondeterministic imperative parallel program using **par**-blocks, **parfor** statements, and communication constructs built on top of **sync** variables and **atomic** functions. Our challenge is, for each problem, to produce a solution that is at least as simple and elegant as the best of the eight previously presented solutions.

# 4 Hamming's Problem (extended)

## 4.1 Problem Description

Given a nonempty set of primes $\{a, b, c, \ldots\}$ and a positive integer $n$, output in increasing order and without duplicates all integers of the form:

$$a^i \times b^j \times c^k \times \ldots \leq n$$

This problem is intended to test a notation's ability to express recursive stream computations and producer/consumer parallelism, and to support dynamic task creation.

## 4.2 Motivation

Our solution to this problem introduces our methodology for functional parallel programming and demonstrates its integration with object-based data abstraction and sequential imperative programming. Our solution is very similar to the solutions presented in Id, Haskell, PCN, Scheme, and Sisal, yet with **sync** and **par** removed and **parfor** replaced by **for**, it is a sequential C++ solution.

## 4.3 Solution Outline

The specification of Hamming's Problem is a functional mapping, as shown in Figure 1. With this kind of problem, our program development methodology is as follows:

- Design the solution as a dataflow network.
- Define data structures for which each component can be written at most once and reading an undefined component suspends.
- Translate the dataflow network into either a sequential program in which data is written before it is read (not an option when the network contains feedback), or a parallel program in which concurrency is controlled by the flow of data at runtime.

Using this methodology, we can develop parallel programs that are almost identical to sequential programs.


Figure 1: Hamming's Problem as a functional mapping.

In our solution to this problem, the dataflow network is a pipeline with one stage for each prime number, as shown in Figure 2. The data structures that connect stages of the pipeline are streams of integers with blocking read operations. We can implement the pipeline as a sequential program, with each stage generating its entire output stream before the next stage is executed, or as a parallel program, with all stages executing concurrently, controlled by the blocking of read operations on streams. Figure 3 shows an example of the pipeline.

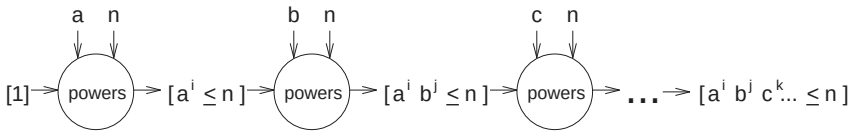


Figure 2: Dataflow pipeline for Hamming's Problem.

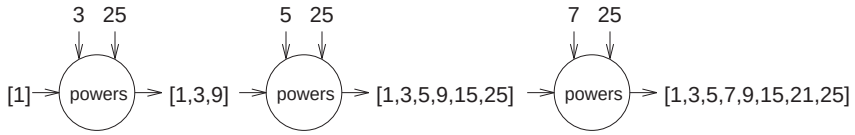


Figure 3: Example of pipeline with primes $\{3, 5, 7\}$ and $n = 25$.

### 4.4 Implementation

Streams of integers are instantiated from the following generic single-writer/single-reader `stream` class:

```
template<class element>
class stream {
public:
    stream(void);                          // Constructor.
    ~stream(void);                         // Destructor.
    void write(const element item);        // Write to rear of stream. (non-blocking)
    void close(void);                      // Terminate writing. (non-blocking)
    void read(element &item);              // Read from front of stream. (blocking)
    element head(void) const;              // Front element of stream. (blocking)
    boolean end_of_stream(void) const; // No more elements to read? (blocking)
private:
    ...
};
```

The single-writer/single-reader restriction combined with the blocking read property ensures determinacy. The `stream` class can be defined as a linked list with **sync** links and mutable front and rear pointers.

The `Hamming` function implements the top level of the program. Input is an array of one or more distinct prime numbers `primes[0..num_primes-1]`, and a positive integer `n`. Output is a stream of integers `result`.

```
void Hamming(const int primes[], const int num_primes,
             const int n, stream<int> &result)
{
    stream<int>* streams = new stream<int>[num_primes];

    par {
        { streams[0].write(1); streams[0].close(); }
        parfor (int i = 0; i < num_primes - 1; i++)
            powers(primes[i], n, streams[i], streams[i + 1]);
        powers(primes[num_primes - 1], n, streams[num_primes - 1], result);
    }
    delete [] streams;
}
```

The stages of the pipeline are executed in parallel, with an array of streams of integers providing the connections between adjacent stages.

The `powers` function implements a single stage of the pipeline. Input is a prime number `prime`, a positive integer `n`, and a stream of integers `input`. Output is a stream of integers `output`.

```
void powers(const int prime, const int n,
            stream<int> &input, stream<int> &output)
{
    int item;
    stream<int> feedback;

    input.read(item);
    output.write(item);
    feedback.write(prime*item);
    do {
        if (!input.end_of_stream() && input.head() < feedback.head())
            input.read(item);
        else
            feedback.read(item);
        if (item <= n) {
            output.write(item);
            feedback.write(prime*item);
        }
    } while (item <= n);
    output.close();
}
```

Each iteration of the sequential loop reads one element from either `input` or `feedback`, and writes one element to both `output` and `feedback`. Execution continues until all `output` elements less than or equal to `n` have been generated. Figure 4 shows an example of the operation of `powers`.

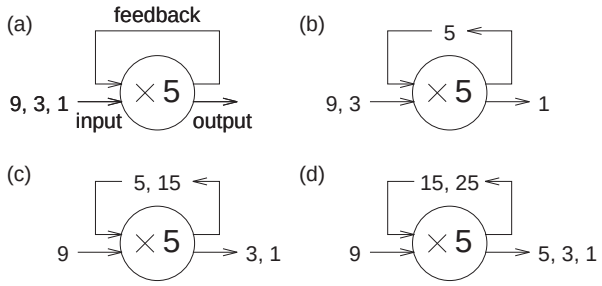


Figure 4: Example of the operation of `powers`.

# 5 The Paraffins Problem

## 5.1 Problem Description

Given an integer $n$, output the chemical structure of all paraffin molecules for $i \leq n$, without repetition and in order of increasing size. The chemical formula for paraffin molecules is $C_iH_{2i+2}$. Include all isomers, but no duplicates. Duplicates are molecules that are identical except for the ordering of bonds. Isomers are molecules that have the same chemical formula but are not duplicates. Figure 5 shows the paraffins of size 1, 2, 3, and 4.

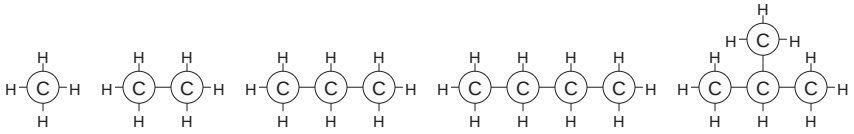

Figure 5: Paraffins of size 1, 2, 3, and 4.

This problem is intended to test a notation's ability to represent, create, and manipulate recursive tree structures, and to express nested loop parallelism.

## 5.2 Motivation

Our solution to this problem presents a second example of our methodology for functional parallel programming and its integration with object-based data abstraction and sequential imperative programming. Again, our solution is very similar to the Id, PCN, Scheme, and Sisal solutions, yet with **sync** and **par** removed and **parfor** replaced by **for**, it is a sequential C++ solution.

## 5.3 Solution Outline

The specification of the Paraffins Problem is a functional mapping. Therefore, we apply the program development methodology described in Section 4.3. In our solution to this problem, the dataflow network is based on the relationship between paraffin molecules and radical molecules. A radical is a molecule with chemical formula $C_iH_{2i+1}$, i.e., a paraffin with one free bond. Figure 6 shows radicals of size 1, 2, and 3.

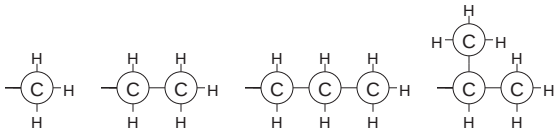

Figure 6: Radicals of size 1, 2, and 3.

A paraffin molecule of size $k$ consists of either a bond center connecting two radicals each of size $k/2$, or a carbon center bonded to four radicals each of size $< k/2$. Therefore, paraffins of size $k$ can be generated by enumerating all distinct quadruples of radicals each of size $< k/2$ with combined size $k-1$, and if $k$ is even, enumerating all distinct pairs of radicals each of size $k/2$.

A radical of size $k > 0$ consists of a carbon atom bonded to three subradicals with combined size $k - 1$. Therefore, radicals of size $k > 0$ can be generated by enumerating all distinct triples of radicals each of size $< k$ with combined size $k - 1$. The hydrogen atom is the only radical of size $k = 0$.

Our solution is to generate lists of radicals of size 0 to $n/2$, and to generate lists of paraffins of size 1 to $n$ from those radicals, as shown in Figure 7. The list data structures have blocking read operations. The solution can be implemented as a sequential program, with radicals generated in increasing size, then paraffins generated in increasing size, or as a parallel program,

with radicals of all sizes and paraffins of all sizes generated concurrently, controlled by the blocking of read operations on lists.

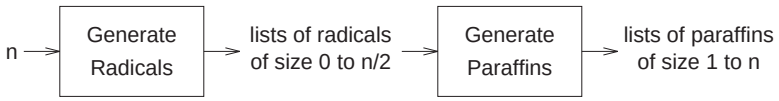


Figure 7: Generating paraffins from radicals.

### 5.4 Implementation

Lists are instantiated from the following generic single-writer `list` class declaration:

```
template<class element>
class list {
public:
    list(void);                          // Constructor.
    ~list(void);                         // Destructor.
    void append(const element item);     // Append to rear of list. (non-blocking)
    void close(void);                    // Terminate appending. (non-blocking)
    pointer<element> open(void) const; // Pointer for reading. (non-blocking)
private:
    ...
};
```

Pointers for reading from lists are instantiated from the following generic single-reader `pointer` class declaration:

```
template<class element>
class pointer {
public:
    pointer(void);                  // Constructor.
    ~pointer(void);                 // Destructor.
    void read(element &item);       // Read from and advance pointer. (blocking)
    boolean end_of_list(void) const; // No more elements to read? (blocking)
    pointer(const pointer<element> &copy_from);
    pointer<element>& operator=(const pointer<element> &assign_from);
    boolean operator==(const pointer<element> right);
private:
    ...
};
```

The single-writer and single-reader restrictions combined with the blocking read property ensure determinacy. The `list` class can be defined as a linked

list with **sync** links and a mutable rear pointer. The `pointer` class can be defined as a mutable pointer.

The `generate_paraffins` function implements the top level of the program. Input is a nonnegative integer `n`. Output is an array of lists of paraffins `paraffins[0..n-1]`, where `paraffins[i]` is a list of paraffins of size `i`+1.

```
void generate_paraffins(const int n, list<paraffin> paraffins[])
{
    list<radical>* radicals = new list<radical>[n/2 + 1];

    par {
        parfor (int rsize = 0; rsize <= n/2; rsize++)
            radicals_of_size(rsize, radicals, radicals[rsize]);
        parfor (int psize = 1; psize <= n; psize++)
            paraffins_of_size(psize, radicals, paraffins[psize - 1]);
    }
    delete [] radicals;
}
```

Lists of radicals of size 0 to `n/2` and lists of paraffins of sizes 1 to `n` are generated in parallel. For brevity, we do not give the declarations of the `radical` and `paraffin` classes. Both can be defined as pointers to dynamically allocated structures.

The `radicals_of_size` function implements the generation of a list of radicals of a given size. Input is a nonnegative integer `size`, and an array of lists of smaller radicals `radicals[0..size-1]`. Output is a list of radicals `result`.

```
void radicals_of_size(const int size, const list<radical> radicals[],
                      list<radical> &result)
{
    if (size == 0)
        result.append(hydrogen_radical());
    else
        for (int k = (size+1)/3; k <= size-1; k++)
            for (int j = (size-k)/2; j <= min(k, size-1-k); j++)
                radicals_of_shape(size-1-k-j, j, k, radicals, result);
    result.close();
}
```

For zero `size`, the only radical is the hydrogen atom. For positive `size`, two nested loops generate all distinct triples of subradical sizes with combined size `size-1`. Radicals of each of these "shapes" are appended to `result`.

The `paraffins_of_size` function implements the generation of a list of paraffins of a given size. Input is a positive integer `size`, and an array of

lists of radicals `radicals[0..size/2-1]`. Output is a list of paraffins `result`.

```
void paraffins_of_size(const int size, const list<radical> radicals[],
                       list<paraffin> &result)
{
    if (size % 2 == 0)
        bond_centered_paraffins(radicals[size/2], result);
    for (int l = (size+2)/4; l <= (size-1)/2; l++)
        for (int k = (size+1-l)/3; k <= min(l, size-1-l); k++)
            for (int j = (size-l-k)/2; j <= min(k, size-1-l-k); j++)
                carbon_centered_paraffins(size-1-l-k-j, j, k, l,
                                          radicals, result);
    result.close();
}
```

For even `size`, bond-centered paraffins consisting of two radicals of size `size/2` are appended to `result`. For all `size`, three nested loops generate all distinct quadruples of radical sizes each $<$`size/2` with combined size `size-1`. Carbon-centered paraffins of each of these "shapes" are appended to `result`.

The `radicals_of_shape` function implements the generation of a sublist of radicals of a given shape. Input is the subradical sizes `i`, `j`, and `k` with $0 \leq$ `i` $\leq$ `j` $\leq$ `k`, and an array of lists of smaller radicals `radicals[0..i+j+k]`. Output is a list of radicals `result`.

```
void radicals_of_shape(const int i, const int j, const int k,
                       const list<radical> radicals[], list<radical> &result)
{
    pointer<radical> pi, pj, pk;
    radical subradical_1, subradical_2, subradical_3;

    pk = radicals[k].open();
    while (!pk.end_of_list()) {
        pk.read(subradical_3);
        pj = radicals[j].open();
        while (!(pj.end_of_list() || (j == k && pj == pk))) {
            pj.read(subradical_2);
            pi = radicals[i].open();
            while (!(pi.end_of_list() || (i == j && pi == pj))) {
                pi.read(subradical_1);
                result.append(carboniferous_radical(
                                  subradical_1, subradical_2, subradical_3));
            }
        }
    }
}
```

Three nested loops generate and append to `result` all distinct radicals with subradicals of sizes `i`, `j`, and `k`. For brevity, we do not give the definitions of the functions `bond_centered_paraffins` and `carbon_centered_paraf-`

`fins` which implement the generation of paraffins of given shapes. These are analogous to the `radicals_of_shape` function.

# 6 The Doctor's Office Problem

## 6.1 Problem Description

Given a set of patients, a set of doctors, and a receptionist, model the following interactions:

- Initially, all patients are well, and all doctors are in a FIFO queue awaiting sick patients.
- At random times, patients become sick and enter a FIFO queue for treatment by one of the doctors.
- The receptionist handles the two queues, assigning patients to doctors in a first-in-first-out manner.
- Once a doctor and patient are paired, the doctor diagnoses the illness and cures the patient in a random amount of time. The patient is then released, and the doctor rejoins the doctors queue to await another patient.

This problem is intended to test a notation's ability to represent a set of concurrent asynchronous processes with circular dependencies. This is neither an event-driven nor time-driven simulation.

## 6.2 Motivation

Our solution to this problem introduces our methodology for imperative parallel programming. Our solution is almost trivial, as a result of being able to represent directly state, concurrency, and nondeterminacy from the problem specification. In the original solutions, functional languages without any source of nondeterminacy were unable to satisfy the specification. Additionally, all solutions based on functional programming were complicated by the need to represent state as an infinite sequence of values. Our solution is most similar to the parallel imperative solution presented in Ada.

### 6.3 Solution Outline

The specification of the Doctor's Office Problem defines explicitly concurrent asynchronous entities and their nondeterministic interactions. With this kind of problem, our program development methodology is as follows:

- Identify the concurrent entities in the system.
- Decide on a model of communication to represent interactions between concurrent entities.
- Identify the state transitions of each concurrent entity as it communicates with other entities.
- Translate the system directly into a parallel program with one process representing each concurrent entity and with concurrency controlled by communication.

In our solution to this problem, the concurrent entities are the patients, the doctors, and the receptionist. We model the interactions between entities as message-passing on FIFO channels. Each patient and each doctor has an input channel to which other entities can send messages. The receptionist has two separate input channels: one to which patients send messages, and one to which doctors send messages. Figure 8 shows the concurrent entities and communication channels. Figures 9, 10, and 11 show the state transitions of patients, doctors, and the receptionist. For simplicity, the system never terminates.

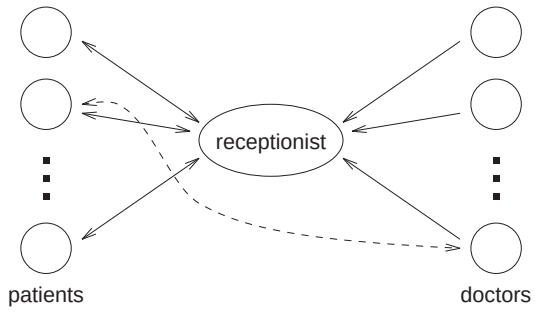


Figure 8: Entities and communication in the Doctor's Office Problem

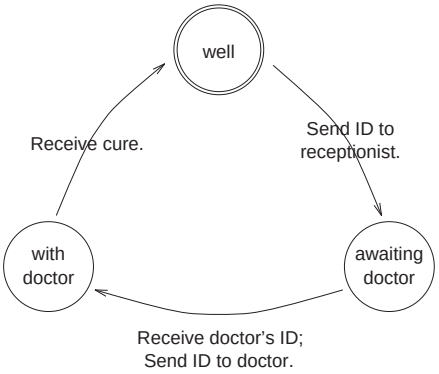


Figure 9: State transitions for patients.

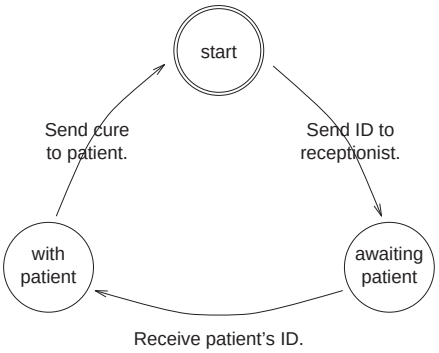


Figure 10: State transitions for doctors.

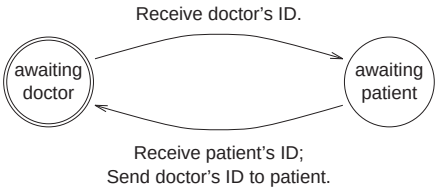


Figure 11: State transitions for the receptionist.

## 6.4 Implementation

Channels are instantiated from the following generic multiple-writer/single-reader FIFO `channel` class:

```
template<class message>
class channel {
public:
    channel(void);                          // Constructor.
    ~channel(void);                         // Destructor.
    atomic void send(const message item); // Non-blocking send.
    void receive(message &item);            // Blocking receive.
private:
    ...
};
```

The `send` function is **atomic** to prevent the interference of multiple concurrent writers. Concurrent `send` operations on the same channel are executed in fair nondeterministic order. The `channel` class can be defined as a linked list with **sync** links and mutable front and rear pointers.

The `office` function implements the top level of the program. Input is the number of patients `num_patients`, and the number of doctors `num_doctors`.

```
void office(const int num_patients, const int num_doctors)
{
    channel<int>* to_patients = new channel<int>[num_patients];
    channel<int>* to_doctors = new channel<int>[num_doctors];
    channel<int> from_patients; // (to receptionist).
    channel<int> from_doctors;  // (to receptionist).

    par {
        parfor (int p = 0; p < num_patients; p++)
            patient(p, to_patients[p], from_patients, to_doctors);
        parfor (int d = 0; d < num_doctors; d++)
            doctor(d, to_doctors[d], from_doctors, to_patients);
        receptionist(from_patients, from_doctors, to_patients);
    }
    delete [] to_patients;
    delete [] to_doctors;
}
```

The given number of patient and doctor processes and the receptionist process are executed in parallel. Channels are declared for communication between processes.

The `patient` function implements a patient process. Input is the patient's identification number `my_ID`, and a channel from other processes `input`. Output is channels to other processes `to_receptionist` and `to_doctors`.

```
void patient(const int my_ID, channel<int> &input,
             channel<int> &to_receptionist, channel<int> to_doctors[])
{
    while (true) {
        int doctor_ID;
        int cure;

        well(my_ID);
        to_receptionist.send(my_ID);
        input.receive(doctor_ID);
        to_doctors[doctor_ID].send(my_ID);
        with_doctor(my_ID, input, to_doctors[doctor_ID]);
        input.receive(cure);
    }
}
```

The implementation follows directly from the state transitions shown in Figure 9. For brevity, we do not give the implementations of the `doctor` and `receptionist` functions. These follow directly from the state transitions shown in Figures 10 and 11 and are analogous to the `patient` function.

# 7 The Skyline Matrix Problem

## 7.1 Problem Description

Solve the following system of linear equations:

$$Ax = b$$

without pivoting, where $A$ is an $n$ by $n$ skyline matrix. A skyline matrix has nonzero values in row $i$ in columns $row_i$ through $i$, and nonzero values in column $j$ in rows $column_j$ through $j$, where $row$ and $column$ are vectors of size $n$. Figure 12 shows an example of a skyline matrix.

This problem is intended to test a notation's ability to represent sparse array structures, and to express efficient parallel and iterative computations on those structures.

## 7.2 Motivation

Our solution to this problem presents a third example of our methodology for functional parallel programming. Object-based data abstraction allows us to develop a program that is independent of the representation of skyline matrices. Again, our solution is very similar to the solutions presented in

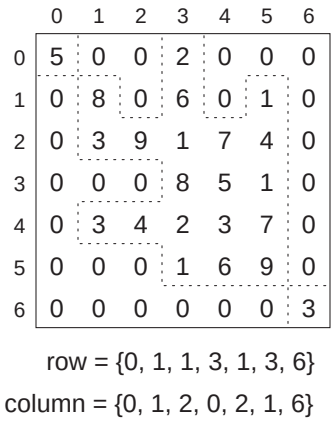


Figure 12: Example of a skyline matrix.

Haskell, Id, PCN, Scheme, and Sisal, yet with **sync** and **par** removed and **parfor** replaced by **for**, our solution is a sequential C++ solution.

### 7.3 Solution Outline

The specification of the Skyline Matrix Problem is a functional mapping. Therefore, we apply the programming methodology described in Section 4.3. In our solution to this problem, the dataflow network is a pipeline consisting of LU factorization, forward substitution, and backward substitution, as shown in Figure 13. At the top level, our design does not depend on the special skyline structure of $A$.

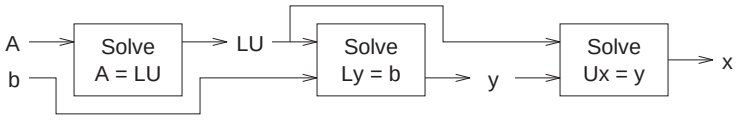


Figure 13: Dataflow pipeline for the Skyline Matrix Problem.

Dataflow within each stage of the pipeline is determined by the following mathematical equations:

$$\text{(a)} \quad l_{ij} = (a_{ij} - \sum_{k=0}^{j-1} l_{ik} \times u_{kj})/u_{jj} \qquad 0 \le i < n,\ 0 \le j \le i$$

$$\text{(b)} \quad u_{ij} = a_{ij} - \sum_{k=0}^{i-1} l_{ik} \times u_{kj} \qquad 0 \le i < n,\ i \le j < n$$

$$\text{(c)} \quad y_i = b_i - \sum_{j=0}^{i-1} l_{ij} \times y_j \qquad 0 \le i < n$$

$$\text{(d)} \quad x_i = (y_i - \sum_{j=i+1}^{n-1} u_{ij} \times x_j)/u_{ii} \qquad 0 \le i < n$$

$L$ is a lower triangular matrix with unit diagonal, and $U$ is a general upper triangular matrix. We represent $L$ and $U$ as a square matrix $LU$, consisting of $U$ and the below-diagonal elements of $L$.

From equations (a) and (b) we see that if $A$ is a skyline matrix, $LU$ is a skyline matrix with the same shape. Therefore, we need only compute elements of $L$ and $U$ that lie inside the skyline. We also see that the range of summation in the innerproduct computations can be reduced to avoid redundant multiplications by zero elements of $L$ and $U$.

The solution can be implemented as a sequential program with the stages of the pipeline executed in sequence, and the output elements of each stage computed in sequence. Alternatively, if we use arrays and vectors of **sync** elements, the solution can be implemented as a parallel program with all stages of the pipeline executed concurrently, and all output elements of each stage computed concurrently, controlled by suspension on undefined **sync** elements. Object-based data abstraction allows the computational structure of the solution to be independent of the particular storage structure used for skyline matrices.

## 7.4 Implementation

Skyline matrices are instantiated from the following generic `matrix` class:

```
template<class element>
class matrix {
public:
    matrix(const int size, const int row[], const int column[]);
    ~matrix(void);
    int size(void) const;
    int* row(void) const;
    int* column(void) const;
    ...
private:
    ...
};
```

The `matrix` class hides the storage structure of skyline matrices from the rest of the program. Skyline matrices are constructed with a given `row[]` and `column[]`, and subscripting of elements outside of this skyline is an error. For brevity, we have omitted the declaration of the subscripting operators. A memory-efficient implementation is two ragged arrays: one for the rows of the lower triangle (excluding the diagonal), and one for the columns of the upper triangle (including the diagonal).

The `solve` function implements the top level of the program. Input is a **sync float** skyline matrix `A`, and a **sync float** vector `b`. Output is a **sync float** vector `x`. `A`, `b`, and `x` must all have the same size. For brevity, we do not give the declaration of the generic `vector` class.

```
void solve(const matrix<sync float> &A, const vector<sync float> &b,
           vector<sync float> &x)
{
    matrix<sync float> LU(A.size(), A.row(), A.column());
    vector<sync float> y(A.size());

    par {
        LU_factorize(A, LU);            // Solve A = LU for L and U.
        forward_substitute(LU, b, y);   // Solve Ly = b for y.
        backward_substitute(LU, y, x); // Solve Ux = y for x.
    }
}
```

The use of **sync** elements allows the LU factorization, forward substitution, and backward substitution stages to be executed in parallel. `LU` is constructed to have the same shape as `A`.

The function `LU_factorize` implements the LU factorization stage of the solution. Input is a **sync float** skyline matrix `A`. Output is a **sync float** skyline matrix `LU`. `A` and `LU` must have the same size and shape.

```
void LU_factorize(const matrix<sync float> &A, matrix<sync float> &LU)
{
    parfor (int i = 0; i < LU.size(); i++) par {
        parfor (int j = LU.row()[i]; j < i; j++) {
            float innerproduct;

            innerproduct = 0.0;
            for (int k = max(LU.row()[i], LU.column()[j]); k < j; k++)
                innerproduct += LU[i][k]*LU[k][j];
            LU[i][j] = (A[i][j] - innerproduct)/LU[j][j];
        }
        parfor (int J = LU.column()[i]; J <= i; J++) {
            float innerproduct;

            innerproduct = 0.0;
            for (int k = max(LU.row()[J], LU.column()[i]); k < J; k++)
                innerproduct += LU[J][k]*LU[k][i];
            LU[J][i] = A[J][i] - innerproduct;
        }
    }
}
```

The implementation follows directly from equations (a) and (b), with loop ranges adjusted to avoid writing or reading matrix elements outside of the skyline. For brevity, we do not give the implementations of the functions `forward_substitute` and `backward_substitute`. These follow directly from equations (c) and (d) and are analogous to the `LU_factorize` function.

# 8 Conclusion

In this report we have presented compatible methodologies for both functional and imperative parallel programming in the context of a popular sequential object-based language extended with a few simple constructs for expressing and controlling parallelism. The concepts that are involved are independent of the particular base language.

Our functional parallel programming methodology is applicable to problems with specifications given as functional mappings. We develop the solution as a dataflow network, then translate the network into either a sequential program or a parallel program with concurrency controlled by the flow of data at runtime. The basic structure of the program is the same in both cases. For the three Salishan Problems of this kind, our parallel solutions were genuinely no more difficult to develop than sequential solutions. In fact, except for the addition of a few keywords, the parallel programs are identical to sequential programs that satisfy the same specifications.

Our imperative parallel programming methodology is applicable to problems with specifications that define explicitly concurrent entities and their interactions. We represent the concurrent entities as parallel processes and their interactions as communication between processes. For the one Salishan Problem of this kind, our parallel solution was almost trivial, as a result of being able to represent concurrency, state, and nondeterminacy directly. Most notations that support functional parallel programming are not well suited to this kind of problem.

The contribution of this work is to show how both these methodologies can be integrated within the framework of a popular sequential object-based language. Integration in this manner complements the strengths of both functional and imperative parallel programming and makes these methodologies accessible to a large body of programming practitioners.

## Acknowledgments

We are grateful to the authors of the original solutions to the Salishan Problems for the quality of their solutions and accompanying discussions. We also thank all the members of the Compositional Systems Group at Caltech for their many discussions and suggestions.

## References

[1] G. R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, Redwood City, California, 1991.

[2] T. Ito and R. H. Halstead, Jr., editors. *Parallel Lisp: Languages and Systems.* Springer-Verlag, Berlin, Germany, 1990. Proceedings of US/Japan Workshop on Parallel Lisp.

[3] B. K. Szymanski, editor. *Parallel Functional Languages and Compilers.* ACM Press, New York, New York, 1991.

[4] D. Cann. Retire Fortran? A debate rekindled. *Communications of the ACM*, 35(8):81–89, August 1992.

[5] R. K. Dybvig. *The Scheme Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[6] K. M. Chandy and S. Taylor. *An Introduction to Parallel Programming.* Jones and Bartlett, Boston, Massachusetts, 1992.

[7] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object oriented programming language. Technical Report CS-TR-92-01, Computer Science Department, California Institute of Technology, 1992.

[8] P. Carlin, M. Chandy, and C. Kesselman. The Compositional C++ language definition. Technical Report CS-TR-92-02, Computer Science Department, California Institute of Technology, 1992.

[9] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, Massachusetts, 1990.

[10] P. Sivilotti. A verified integration of imperative parallel programming paradigms in an object-oriented language. Technical Report CS-TR-93-21, Computer Science Department, California Institute of Technology, 1993.

[11] J. T. Feo, editor. *A Comparative Study of Parallel Programming Languages: The Salishan Problems*, volume 6 of *Special Topics in Supercomputing.* North-Holland, Amsterdam, The Netherlands, 1992.

# A Hamming's Problem (extended)

## A.1 streams.h

```
#ifndef STREAMS
#define STREAMS
/*****************************************************************************/
/*                                                                           */
/*                    Stream Class Template Declaration                      */
/*                    ---------------------------------                      */
/*                                                                           */
/* Written by: John Thornley, Computer Science Dept., Caltech.               */
/* Last modified: Tuesday 30th November 1993.                                */
/*                                                                           */
/* DESCRIPTION:                                                              */
/*                                                                           */
/* Generic single-writer/single-reader stream class:                         */
/*                                                                           */
/* - Non-blocking write operations: write and close.                         */
/* - Blocking read operations: read, head, and end_of_stream.                */
/* - Copy and assignment operations are prohibited.                          */
/*                                                                           */
/* Multiple concurrent write operations are erroneous, multiple concurrent   */
/* read operations are erroneous, but concurrent write and read operations   */
/* are allowed.                                                              */
/*                                                                           */
/*****************************************************************************/

#include "boolean.h"

template<class element>
struct node;

template<class element>
class stream {
public:
    stream(void);
    ~stream(void);

    void write(const element item);    // Non-blocking.
    void close(void);                  // Non-blocking.

    void read(element &item);          // Blocking.
    element head(void) const;          // Blocking.
    boolean end_of_stream(void) const; // Blocking.
private:
    node<element>* front;
    node<element>* rear;
    boolean closed;

    stream(const stream<element> &copy_from);
    stream<element>& operator=(const stream<element> &assign_from);
};

/*****************************************************************************/
```

```
#endif // STREAMS
```

## A.2 streams.C

```
/*************************************************************************/
/*                                                                       */
/*                     Stream Class Template Definition                  */
/*                     --------------------------------                  */
/*                                                                       */
/* Written by: John Thornley, Computer Science Dept., Caltech.           */
/* Last modified: Tuesday 30th November 1993.                            */
/*                                                                       */
/*************************************************************************/

#include <stdio.h>
#include <assert.h>
#include "boolean.h"
#include "streams.h"

//-------------------------------------------------------------------------

template<class element>
struct node {
    element Item;
    node *sync next;
    node(void) { }
    node(const element item) { Item = item; }
};

//-------------------------------------------------------------------------

template<class element>
stream<element>::stream(void)
{
    front = new node<element>;
    assert(front != NULL);
    rear = front;
    closed = false;
}

//-------------------------------------------------------------------------

template<class element>
stream<element>::~stream(void)
{
    while (front != rear) {
        node<element>* old_front;

        old_front = front;
        front = front->next;
        delete old_front;
    }
    delete front;
}

//-------------------------------------------------------------------------
```

```
template<class element>
void stream<element>::write(const element item)
{
    assert(!closed);
    node<element>* new_rear;

    new_rear = new node<element>(item);
    assert(new_rear != NULL);
    rear->next = new_rear;
    rear = new_rear;
}

//---------------------------------------------------------------------------

template<class element>
void stream<element>::close(void)
{
    assert(!closed);
    rear->next = NULL;
    closed = true;
}

//---------------------------------------------------------------------------

template<class element>
void stream<element>::read(element &item)
{
    assert(front->next != NULL);
    node<element>* old_front;

    old_front = front;
    front = front->next;
    delete old_front;
    item = front->Item;
}

//---------------------------------------------------------------------------

template<class element>
element stream<element>::head(void) const
{
    assert(front->next != NULL);
    return front->next->Item;
}

//---------------------------------------------------------------------------

template<class element>
boolean stream<element>::end_of_stream(void) const
{
    return front->next == NULL;
}

/*****************************************************************************/
```

## A.3 Hamming.h

```
#ifndef HAMMING
#define HAMMING
/****************************************************************************/
/*                                                                          */
/*                      Hamming Function Declaration                        */
/*                      ----------------------------                        */
/*                                                                          */
/* Written by: John Thornley, Computer Science Dept., Caltech.              */
/* Last modified: Tuesday 30th November 1993.                               */
/*                                                                          */
/* DESCRIPTION:                                                             */
/*                                                                          */
/* Given a set of prime numbers {a, b, c, ...} and an integer n, generate   */
/* in increasing order and without duplicates, a stream of integers of the  */
/* form:                                                                    */
/*                                                                          */
/*                          i    j    k                                     */
/*                         a  * b  * c  * ... <= n.                         */
/*                                                                          */
/* PROBLEM FROM:                                                            */
/*                                                                          */
/* "A Comparative Study of Parallel Programming Languages: The Salishan     */
/* Problems", edited by John T. Feo.  Special Topics in Supercomputing, Vol-*/
/* ume 6.  North-Holland, Amsterdam, 1992.                                  */
/*                                                                          */
/****************************************************************************/

#include "streams.h"

void Hamming(const int primes[], const int num_primes,
             const int n, stream<int> &result);
// Input Condition:
//     num_primes > 0 and
//     primes[0] .. primes[num_primes - 1] are distinct prime numbers and
//     n > 0 and
//     result is an empty and not closed stream.
// Output Condition:
//     result is a closed stream, in increasing order and without duplicates,
//     of all integers of the form:
//                     i           j                              x
//           primes[0] * primes[1] * ... * primes[num_primes - 1]  <= n.

/****************************************************************************/
#endif // HAMMING
```

## A.4 Hamming.C

```
/*****************************************************************************/
/*                                                                           */
/*                        Hamming Function Definition                        */
/*                        ---------------------------                        */
/*                                                                           */
/* Written by: John Thornley, Computer Science Dept., Caltech.               */
/* Last modified: Tuesday 30th November 1993.                                */
/*                                                                           */
/*****************************************************************************/

#include "streams.h"
#include "Hamming.h"

//---------------------------------------------------------------------------

void powers(const int prime, const int n,
            stream<int> &input, stream<int> &output)
{
    int item;
    stream<int> feedback;

    input.read(item);
    output.write(item);
    feedback.write(prime*item);

    do {
        if (!input.end_of_stream() && input.head() < feedback.head())
            input.read(item);
        else
            feedback.read(item);
        if (item <= n) {
            output.write(item);
            feedback.write(prime*item);
        }
    } while (item <= n);

    output.close();
}

//---------------------------------------------------------------------------

void Hamming(const int primes[], const int num_primes,
             const int n, stream<int> &result)
{
    stream<int>* streams = new stream<int>[num_primes];

    par {
        { streams[0].write(1); streams[0].close(); }
        parfor (int i = 0; i < num_primes - 1; i++)
            powers(primes[i], n, streams[i], streams[i + 1]);
        powers(primes[num_primes - 1], n, streams[num_primes - 1], result);
    }
    delete [] streams;
```

```
}

/*****************************************************************************/
```

# B The Paraffins Problem

## B.1 radicals.h

```
#ifndef RADICALS
#define RADICALS
/*****************************************************************************/
/*                                                                           */
/*                        Radical Class Declaration                          */
/*                        -------------------------                          */
/*                                                                           */
/* Written by: John Thornley, Computer Science Dept., Caltech.               */
/* Last modified: Tuesday 30th November 1993.                                */
/*                                                                           */
/* DESCRIPTION:                                                              */
/*                                                                           */
/* Radical molecule class.  A radical molecule consists of either a single   */
/* hydrogen atom or a carbon atom bonded to three smaller radical molecules. */
/* cules.                                                                    */
/*                                                                           */
/*****************************************************************************/

struct radical_descriptor;

typedef enum {hydrogen, carboniferous} radical_kind;

class radical {
public:
    radical(void);
    ~radical(void);

    friend radical hydrogen_radical(void);
    friend radical carboniferous_radical(const radical subradical_1,
                                         const radical subradical_2,
                                         const radical subradical_3 );

    radical_kind kind(void) const;
    radical subradical(const int position) const;

    radical(const radical &copy_from);
    radical& operator=(const radical &assign_from);
private:
    radical_descriptor* access;
};

/*****************************************************************************/
#endif // RADICALS
```

## B.2 radicals.C

```
/*****************************************************************************/
/*                                                                           */
/*                         Radical Class Definition                          */
/*                         ------------------------                          */
/*                                                                           */
/* Written by: John Thornley, Computer Science Dept., Caltech.               */
/* Last modified: Tuesday 30th November 1993.                                */
/*                                                                           */
/*****************************************************************************/

#include <stdio.h>
#include <assert.h>
#include "radicals.h"

//---------------------------------------------------------------------------

struct radical_descriptor {
    radical_kind kind;
    radical subradicals[3];
};

//---------------------------------------------------------------------------

radical::radical(void)
{
    access = NULL;
}

//---------------------------------------------------------------------------

radical::~radical(void)
{
}

//---------------------------------------------------------------------------

radical hydrogen_radical(void)
{
    radical result;

    result.access = new radical_descriptor;
    assert(result.access != NULL);
    result.access->kind = hydrogen;
    return result;
}

//---------------------------------------------------------------------------

radical carboniferous_radical(const radical subradical_1,
                              const radical subradical_2,
                              const radical subradical_3 )
{
    radical result;
```

```
    result.access = new radical_descriptor;
    assert(result.access != NULL);
    result.access->kind = carboniferous;
    result.access->subradicals[0] = subradical_1;
    result.access->subradicals[1] = subradical_2;
    result.access->subradicals[2] = subradical_3;
    return result;
}

//------------------------------------------------------------------------------

radical_kind radical::kind(void) const
{
    assert(access != NULL);
    return access->kind;
}

//------------------------------------------------------------------------------

radical radical::subradical(const int position) const
{
    assert(access != NULL);
    assert(access->kind == carboniferous);
    assert(position == 1 || position == 2 || position == 3);
    return access->subradicals[position - 1];
}

//------------------------------------------------------------------------------

radical::radical(const radical& copy_from)
{
    assert(copy_from.access != NULL);
    access = copy_from.access;
}

//------------------------------------------------------------------------------

radical& radical::operator=(const radical& assign_from)
{
    assert(assign_from.access != NULL);
    access = assign_from.access;
    return *this;
}

/******************************************************************************/
```

## B.3 paraffins.h

```
#ifndef PARAFFINS
#define PARAFFINS
/*****************************************************************************/
/*                                                                           */
/*                        Paraffin Class Declaration                         */
/*                        --------------------------                         */
/*                                                                           */
/* Written by: John Thornley, Computer Science Dept., Caltech.               */
/* Last modified: Tuesday 30th November 1993.                                */
/*                                                                           */
/* DESCRIPTION:                                                              */
/*                                                                           */
/* Paraffin molecule class.  A paraffin molecule consists of either a bond   */
/* center connecting two radical molecules or a carbon center bonded to four */
/* radical molecules.                                                        */
/*                                                                           */
/*****************************************************************************/

#include "radicals.h"

struct paraffin_descriptor;

typedef enum {bond_centered, carbon_centered} paraffin_kind;

class paraffin {
public:
    paraffin(void);
    ~paraffin(void);

    friend paraffin bond_centered_paraffin(const radical neighbor_1,
                                           const radical neighbor_2 );
    friend paraffin carbon_centered_paraffin(const radical neighbor_1,
                                             const radical neighbor_2,
                                             const radical neighbor_3,
                                             const radical neighbor_4 );

    paraffin_kind kind(void) const;
    radical bond_neighbor(const int position) const;
    radical carbon_neighbor(const int position) const;

    paraffin(const paraffin &copy_from);
    paraffin& operator=(const paraffin &assign_from);
private:
    paraffin_descriptor* access;
};

/*****************************************************************************/
#endif // PARAFFINS
```

## B.4 paraffins.C

```
/*****************************************************************************/
/*                                                                           */
/*                        Paraffin Class Definition                          */
/*                        -------------------------                          */
/*                                                                           */
/* Written by: John Thornley, Computer Science Dept., Caltech.               */
/* Last modified: Tuesday 30th November 1993.                                */
/*                                                                           */
/*****************************************************************************/

#include <stdio.h>
#include <assert.h>
#include "radicals.h"
#include "paraffins.h"

//---------------------------------------------------------------------------

struct paraffin_descriptor {
    paraffin_kind kind;
    radical       neighbors[4];
};

//---------------------------------------------------------------------------

paraffin::paraffin(void)
{
    access = NULL;
}

//---------------------------------------------------------------------------

paraffin::~paraffin(void)
{
}

//---------------------------------------------------------------------------

paraffin bond_centered_paraffin(const radical neighbor_1,
                                const radical neighbor_2 )
{
    paraffin result;

    result.access = new paraffin_descriptor;
    assert(result.access != NULL);
    result.access->kind = bond_centered;
    result.access->neighbors[0] = neighbor_1;
    result.access->neighbors[1] = neighbor_2;
    return result;
}

//---------------------------------------------------------------------------

paraffin carbon_centered_paraffin(const radical neighbor_1,
```

```
                                const radical neighbor_2,
                                const radical neighbor_3,
                                const radical neighbor_4 )
{
    paraffin result;

    result.access = new paraffin_descriptor;
    assert(result.access != NULL);
    result.access->kind = carbon_centered;
    result.access->neighbors[0] = neighbor_1;
    result.access->neighbors[1] = neighbor_2;
    result.access->neighbors[2] = neighbor_3;
    result.access->neighbors[3] = neighbor_4;
    return result;
}

//------------------------------------------------------------------------------

paraffin_kind paraffin::kind(void) const
{
    assert(access != NULL);
    return access->kind;
}

//------------------------------------------------------------------------------

radical paraffin::bond_neighbor(const int position) const
{
    assert(access != NULL);
    assert(access->kind == bond_centered);
    assert(position == 1 || position == 2);
    return access->neighbors[position - 1];
}

//------------------------------------------------------------------------------

radical paraffin::carbon_neighbor(const int position) const
{
    assert(access != NULL);
    assert(access->kind == carbon_centered);
    assert(position == 1 || position == 2 ||
           position == 3 || position == 4   );
    return access->neighbors[position - 1];
}

//------------------------------------------------------------------------------

paraffin::paraffin(const paraffin &copy_from)
{
    assert(copy_from.access != NULL);
    access = copy_from.access;
}

//------------------------------------------------------------------------------
```

```
paraffin& paraffin::operator=(const paraffin &assign_from)
{
    assert(assign_from.access != NULL);
    access = assign_from.access;
    return *this;
}

/*****************************************************************************/
```

## B.5 lists.h

```
#ifndef LISTS
#define LISTS
/*************************************************************************/
/*                                                                       */
/*               List and Pointer Class Template Declarations            */
/*               --------------------------------------------            */
/*                                                                       */
/* Written by: John Thornley, Computer Science Dept., Caltech.           */
/* Last modified: Tuesday 30th November 1993.                            */
/*                                                                       */
/* DESCRIPTION:                                                          */
/*                                                                       */
/* Generic single-writer list class:                                     */
/*                                                                       */
/* - Non-blocking write operations: append and close.                     */
/* - Non-blocking open (for reading) operation.                          */
/* - Copy and assignment operations are prohibited.                      */
/*                                                                       */
/* Multiple concurrent write operations are erroneous.                   */
/*                                                                       */
/* Generic single-reader pointer class:                                  */
/*                                                                       */
/* - Blocking read operations: read and end_of_list.                     */
/* - Copy, assignment, and equality operations are defined.              */
/*                                                                       */
/* Multiple concurrent read operations are erroneous.                    */
/*                                                                       */
/*************************************************************************/

#include "boolean.h"

template<class element>
struct node;

template<class element>
class list;

template<class element>
class pointer {
public:
    pointer(void);
    ~pointer(void);

    void read(element &item);        // Blocking.
    boolean end_of_list(void) const; // Blocking.

    pointer(const pointer<element> &copy_from);
    pointer<element>& operator=(const pointer<element> &assign_from);
    boolean operator==(const pointer<element> right);
private:
    node<element>* Position;
    pointer(node<element>* position);
```

```
    friend class list<element>;
};

template<class element>
class list {
public:
    list(void);
    ~list(void);

    void append(const element item);   // Non-blocking.
    void close(void);                  // Non-blocking.
    pointer<element> open(void) const; // Non-blocking.
private:
    node<element>* head;
    node<element>* tail;
    boolean closed;

    list(const list<element> &copy_from);
    list<element>& operator=(const list<element> &assign_from);
};

/*****************************************************************************/
#endif // LISTS
```

## B.6 lists.C

```
/*************************************************************************/
/*                                                                       */
/*                List and Pointer Class Template Definitions            */
/*                -------------------------------------------            */
/*                                                                       */
/* Written by: John Thornley, Computer Science Dept., Caltech.           */
/* Last modified: Tuesday 30th November 1993.                            */
/*                                                                       */
/*************************************************************************/

#include <stdio.h>
#include <assert.h>
#include "boolean.h"
#include "lists.h"

//-------------------------------------------------------------------------

template<class element>
struct node {
    element Item;
    node *sync next;
    node(void) { }
    node(element item) { Item = item; }
};

//-------------------------------------------------------------------------

template<class element>
pointer<element>::pointer(void)
{
    Position = NULL;
}

//-------------------------------------------------------------------------

template<class element>
pointer<element>::pointer(node<element>* position)
{
    Position = position;
}

//-------------------------------------------------------------------------

template<class element>
pointer<element>::~pointer(void)
{
}

//-------------------------------------------------------------------------

template<class element>
void pointer<element>::read(element &item)
{
```

```
    assert(Position != NULL && Position->next != NULL);
    Position = Position->next;
    item = Position->Item;
}

//---------------------------------------------------------------------------

template<class element>
boolean pointer<element>::end_of_list(void) const
{
    assert(Position != NULL);
    return Position->next == NULL;
}

//---------------------------------------------------------------------------

template<class element>
pointer<element>::pointer(const pointer<element> &copy_from)
{
    assert(copy_from.Position != NULL);
    Position = copy_from.Position;
}

//---------------------------------------------------------------------------

template<class element>
pointer<element>&
pointer<element>::operator=(const pointer<element> &assign_from)
{
    assert(assign_from.Position != NULL);
    Position = assign_from.Position;
    return *this;
}

//---------------------------------------------------------------------------

template<class element>
boolean pointer<element>::operator==(const pointer<element> right)
{
    assert(Position != NULL && right.Position != NULL);
    return Position == right.Position;
}

//---------------------------------------------------------------------------

template<class element>
list<element>::list(void)
{
    head = new node<element>;
    assert(head != NULL);
    tail = head;
    closed = false;
}

//---------------------------------------------------------------------------
```

```
template<class element>
list<element>::~list(void)
{
    while (head != tail) {
        node<element>* old_head;

        old_head = head;
        head = head->next;
        delete old_head;
    }
    delete head;
}

//----------------------------------------------------------------------------

template<class element>
void list<element>::append(const element item)
{
    assert(!closed);
    tail->next = new node<element>(item);
    assert(tail->next != NULL);
    tail = tail->next;
}

//----------------------------------------------------------------------------

template<class element>
void list<element>::close(void)
{
    assert(!closed);
    tail->next = NULL;
    closed = true;
}

//----------------------------------------------------------------------------

template<class element>
pointer<element> list<element>::open(void) const
{
    return pointer<element>(head);
}

/****************************************************************************/
```

## B.7 generate.h

```
#ifndef GENERATE
#define GENERATE
/*****************************************************************************/
/*                                                                           */
/*                  Generate_Paraffins Function Declaration                  */
/*                  ---------------------------------------                  */
/*                                                                           */
/* Written by: John Thornley, Computer Science Dept., Caltech.               */
/* Last modified: Tuesday 30th November 1993.                                */
/*                                                                           */
/* DESCRIPTION:                                                              */
/*                                                                           */
/* Given an integer n, generate the chemical structure of all paraffin       */
/* molecules for i <= n.  The chemical formula for paraffin molecules is:    */
/*                                                                           */
/*                                 C H                                       */
/*                                  i 2i+2                                   */
/*                                                                           */
/* Include all isomers, but no duplicates.                                   */
/*                                                                           */
/* PROBLEM FROM:                                                             */
/*                                                                           */
/* "A Comparative Study of Parallel Programming Languages: The Salishan      */
/* Problems", edited by John T. Feo.  Special Topics in Supercomputing, Vol- */
/* Volume 6.  North-Holland, Amsterdam, 1992.                                */
/*                                                                           */
/*****************************************************************************/

#include "paraffins.h"
#include "lists.h"

void generate_paraffins(const int n, list<paraffin> paraffins[]);
// Input Condition:
//     n >= 0 and
//     paraffins[0] .. paraffins[n - 1] are empty and not closed lists.
// Output Condition:
//     For all i in 0 .. n - 1 : paraffins[i] is a closed list of
//         the chemical structures of paraffin molecules of size i + 1.

/*****************************************************************************/
#endif // GENERATE
```

## B.8 generate.C

```
/*************************************************************************/
/*                                                                       */
/*                  Generate_Paraffins Function Definition               */
/*                  --------------------------------------               */
/*                                                                       */
/* Written by: John Thornley, Computer Science Dept., Caltech.           */
/* Last modified: Tuesday 30th November 1993.                            */
/*                                                                       */
/*************************************************************************/

#include "radicals.h"
#include "paraffins.h"
#include "lists.h"
#include "generate.h"

//-------------------------------------------------------------------------

int min(const int x, const int y)
{
    if (x <= y) return x; else return y;
}

//-------------------------------------------------------------------------

void radicals_of_shape(const int i, const int j, const int k,
                       const list<radical> radicals[], list<radical> &result)
{
    pointer<radical> pi, pj, pk;
    radical subradical_1, subradical_2, subradical_3;

    pk = radicals[k].open();
    while (!pk.end_of_list()) {
        pk.read(subradical_3);
        pj = radicals[j].open();
        while (!(pj.end_of_list() || (j == k && pj == pk))) {
            pj.read(subradical_2);
            pi = radicals[i].open();
            while (!(pi.end_of_list() || (i == j && pi == pj))) {
                pi.read(subradical_1);
                result.append(carboniferous_radical(
                                subradical_1, subradical_2, subradical_3));
            }
        }
    }
}

//-------------------------------------------------------------------------

void radicals_of_size(const int size, const list<radical> radicals[],
                      list<radical> &result)
{
    if (size == 0)
        result.append(hydrogen_radical());
```

```
    else
        for (int k = (size+1)/3; k <= size-1; k++)
            for (int j = (size-k)/2; j <= min(k, size-1-k); j++)
                radicals_of_shape(size-1-k-j, j, k, radicals, result);
    result.close();
}

//----------------------------------------------------------------------------

void bond_centered_paraffins(const list<radical> &neighbors,
                             list<paraffin> &result)
{
    pointer<radical> p1, p2;
    radical neighbor_1, neighbor_2;

    p2 = neighbors.open();
    while (!p2.end_of_list()) {
        p2.read(neighbor_2);
        p1 = neighbors.open();
        while (!(p1.end_of_list() || p1 == p2)) {
            p1.read(neighbor_1);
            result.append(bond_centered_paraffin(neighbor_1, neighbor_2));
        }
    }
}

//----------------------------------------------------------------------------

void carbon_centered_paraffins(
         const int i, const int j, const int k, const int l,
         const list<radical> radicals[], list<paraffin> &result)
{
    pointer<radical> pi, pj, pk, pl;
    radical neighbor_1, neighbor_2, neighbor_3, neighbor_4;

    pl = radicals[l].open();
    while (!pl.end_of_list()) {
        pl.read(neighbor_4);
        pk = radicals[k].open();
        while (!(pk.end_of_list() || (k == l && pk == pl))) {
            pk.read(neighbor_3);
            pj = radicals[j].open();
            while (!(pj.end_of_list() || (j == k && pj == pk))) {
                pj.read(neighbor_2);
                pi = radicals[i].open();
                while (!(pi.end_of_list() || (i == j && pi == pj))) {
                    pi.read(neighbor_1);
                    result.append(carbon_centered_paraffin(neighbor_1,
                                    neighbor_2, neighbor_3, neighbor_4));
                }
            }
        }
    }
}
```

```
//-----------------------------------------------------------------------------

void paraffins_of_size(const int size, const list<radical> radicals[],
                       list<paraffin> &result)
{
    if (size % 2 == 0)
        bond_centered_paraffins(radicals[size/2], result);
    for (int l = (size+2)/4; l <= (size-1)/2; l++)
        for (int k = (size+1-l)/3; k <= min(l, size-1-l); k++)
            for (int j = (size-l-k)/2; j <= min(k, size-1-l-k); j++)
                carbon_centered_paraffins(size-1-l-k-j, j, k, l,
                                          radicals, result);
    result.close();
}

//-----------------------------------------------------------------------------

void generate_paraffins(const int n, list<paraffin> paraffins[])
{
    list<radical>* radicals = new list<radical>[n/2 + 1];

    par {
        parfor (int rsize = 0; rsize <= n/2; rsize++)
            radicals_of_size(rsize, radicals, radicals[rsize]);
        parfor (int psize = 1; psize <= n; psize++)
            paraffins_of_size(psize, radicals, paraffins[psize - 1]);
    }
    delete [] radicals;
}

/*****************************************************************************/
```

# C The Doctor's Office Problem

## C.1 channels.h

```
#ifndef CHANNELS
#define CHANNELS
/*************************************************************************/
/*                                                                       */
/*                    Channel Class Template Declaration                 */
/*                    ----------------------------------                 */
/*                                                                       */
/* Written by: John Thornley, Computer Science Dept., Caltech.           */
/* Last modified: Tuesday 30th November 1993.                            */
/*                                                                       */
/* DESCRIPTION:                                                          */
/*                                                                       */
/* Generic multiple-writer/single-reader channel class:                  */
/*                                                                       */
/* - Non-blocking write operation: send.                                 */
/* - Blocking read operation: receive.                                   */
/* - Copy and assignment operations are prohibited.                      */
/*                                                                       */
/* Multiple concurrent write operations are allowed, multiple concurrent */
/* read operations are erroneous, and concurrent write and read operations */
/* are allowed.                                                          */
/*                                                                       */
/*************************************************************************/

template<class element>
struct node;

template<class message>
class channel {
public:
    channel(void);
    ~channel(void);

    atomic void send(const message item); // Non-blocking.
    void receive(message &item);          // Blocking.
private:
    node<message>* front;
    node<message>* rear;

    channel(const channel<message> &copy_from);
    channel<message>& operator=(const channel<message> &assign_from);
};

/*************************************************************************/
#endif // CHANNELS
```

## C.2 channels.C

```
/*************************************************************************/
/*                                                                       */
/*                    Channel Class Template Definition                  */
/*                    ---------------------------------                  */
/*                                                                       */
/* Written by: John Thornley, Computer Science Dept., Caltech.           */
/* Last modified: Tuesday 30th November 1993.                            */
/*                                                                       */
/*************************************************************************/

#include <stdio.h>
#include <assert.h>
#include "channels.h"

//-------------------------------------------------------------------------

template<class element>
struct node {
    element Item;
    node *sync next;
    node(void) { }
    node(const element item) { Item = item; }
};

//-------------------------------------------------------------------------

template<class message>
channel<message>::channel(void)
{
    front = new node<message>;
    assert(front != NULL);
    rear = front;
}

//-------------------------------------------------------------------------

template<class message>
channel<message>::~channel(void)
{
    while (front != rear) {
        node<message>* old_front;

        old_front = front;
        front = front->next;
        delete old_front;
    }
    delete front;
}

//-------------------------------------------------------------------------

template<class message>
atomic void channel<message>::send(const message item)
```

```
{
    node<message>* new_rear;

    new_rear = new node<message>(item);
    assert(new_rear != NULL);
    rear->next = new_rear;
    rear = new_rear;
}

//---------------------------------------------------------------------------

template<class message>
void channel<message>::receive(message &item)
{
    node<message>* old_front;

    old_front = front;
    front = front->next;
    delete old_front;
    item = front->Item;
}

/*****************************************************************************/
```

## C.3 office.h

```
#ifndef OFFICE
#define OFFICE
/****************************************************************************/
/*                                                                          */
/*                        Office Function Declaration                       */
/*                        ---------------------------                       */
/*                                                                          */
/* Written by: John Thornley, Computer Science Dept., Caltech.              */
/* Last modified: Tuesday 30th November 1993.                               */
/*                                                                          */
/* DESCRIPTION:                                                             */
/*                                                                          */
/* Given a set of patients, a set of doctors, and a receptionist, model the */
/* following interactions:                                                  */
/*                                                                          */
/* - Initially, all patients are well, and all doctors are in a FIFO queue  */
/*   awaiting patients.                                                     */
/* - At random times, patients become sick and enter a FIFO queue for treat-*/
/*   ment by one of the doctors.                                            */
/* - The receptionist handles the two queues, assigning patients to doctors */
/*   in a first-in-first-out manner.                                        */
/* - Once a doctor and patient are paired, the doctor diagnoses the illness */
/*   and cures the patient in a random amount of time.  The patient is then */
/*   released, and the doctor rejoins the doctors queue to await another    */
/*   patient.                                                               */
/*                                                                          */
/* No termination conditions are specified.                                 */
/*                                                                          */
/* PROBLEM FROM:                                                            */
/*                                                                          */
/* "A Comparative Study of Parallel Programming Languages: The Salishan     */
/* Problems", edited by John T. Feo.  Special Topics in Supercomputing, Vol-*/
/* ume 6.  North-Holland, Amsterdam, 1992.                                  */
/*                                                                          */
/****************************************************************************/

#include <iostream.h>

void office(const int num_patients, const int num_doctors);
// Input Condition:
//     num_patients > 0 and num_doctors > 0.
// Output:
//     A trace of treatments is written to cout.
//     Execution is non-terminating.

/****************************************************************************/
#endif // OFFICE
```

## C.4 office.C

```
/*************************************************************************/
/*                                                                       */
/*                        Office Function Definition                     */
/*                        --------------------------                     */
/*                                                                       */
/* Written by: John Thornley, Computer Science Dept., Caltech.           */
/* Date of last alteration: Tuesday 30th November 1993.                  */
/*                                                                       */
/*************************************************************************/

#include <iostream.h>
#include "boolean.h"
#include "channels.h"
#include "office.h"

//-------------------------------------------------------------------------

void well(const int my_ID)
{
}

void with_doctor(const int my_ID,
                 channel<int> &from_doctor, channel<int> &to_doctor)
{
}

void patient(const int my_ID, channel<int> &input,
             channel<int> &to_receptionist, channel<int> to_doctors[])
{
    while (true) {
        int doctor_ID;
        int cure;

        well(my_ID);
        to_receptionist.send(my_ID);
        input.receive(doctor_ID);
        to_doctors[doctor_ID].send(my_ID);
        with_doctor(my_ID, input, to_doctors[doctor_ID]);
        input.receive(cure);
    }
}

//-------------------------------------------------------------------------

void with_patient(const int my_ID,
                  channel<int> &from_patient, channel<int> &to_patient)
{
}

void doctor(const int my_ID, channel<int> &input,
            channel<int> &to_receptionist, channel<int> to_patients[])
{
    while (true) {
```

```
        int patient_ID;
        const int cure = 0;

        to_receptionist.send(my_ID);
        input.receive(patient_ID);
        with_patient(my_ID, input, to_patients[patient_ID]);
        to_patients[patient_ID].send(cure);
    }
}

//----------------------------------------------------------------------------

void receptionist(channel<int> &from_patients, channel<int> &from_doctors,
                  channel<int> to_patients[])
{
    int doctor_ID, patient_ID;

    while (true) {
        from_doctors.receive(doctor_ID);
        from_patients.receive(patient_ID);
        cout << "Patient " << patient_ID << " treated by "
             << "Doctor " << doctor_ID << endl;
        to_patients[patient_ID].send(doctor_ID);
    }
}

//----------------------------------------------------------------------------

void office(const int num_patients, const int num_doctors)
{
    channel<int>* to_patients = new channel<int>[num_patients];
    channel<int>* to_doctors = new channel<int>[num_doctors];
    channel<int> from_patients;
    channel<int> from_doctors;

    par {
        parfor (int p = 0; p < num_patients; p++)
            patient(p, to_patients[p], from_patients, to_doctors);
        parfor (int d = 0; d < num_doctors; d++)
            doctor(d, to_doctors[d], from_doctors, to_patients);
        receptionist(from_patients, from_doctors, to_patients);
    }
    delete [] to_patients;
    delete [] to_doctors;
}

/****************************************************************************/
```

# D The Skyline Matrix Problem

## D.1 vectors.h

```
#ifndef VECTORS
#define VECTORS
/*************************************************************************/
/*                                                                       */
/*                    Vector Class Template Declaration                  */
/*                    ---------------------------------                  */
/*                                                                       */
/* Written by: John Thornley, Computer Science Dept., Caltech.           */
/* Last modified: Tuesday 30th November 1993.                            */
/*                                                                       */
/* Generic vector[0..size-1] class:                                      */
/*                                                                       */
/* - Size is an argument of the constructor.                             */
/* - Index values must lie in the range 0..size-1.                       */
/* - Copy and assignment operations are prohibited.                      */
/*                                                                       */
/*************************************************************************/

template<class element>
class vector {
public:
    vector(const int size);
    ~vector(void);

    int size(void) const;
    element& operator[](const int index) const;
private:
    int Size;
    element* Elements;

    vector(void);
    vector(const vector<element> &copy_from);
    vector<element>& operator=(const vector<element> &assign_from);
};

/*************************************************************************/
#endif // VECTORS
```

## D.2 vectors.C

```
/*************************************************************************/
/*                                                                       */
/*                    Vector Class Template Definition                   */
/*                    --------------------------------                   */
/*                                                                       */
/* Written by: John Thornley, Computer Science Dept., Caltech.           */
/* Last modified: Tuesday 30th November 1993.                            */
/*                                                                       */
/*************************************************************************/

#include <stdio.h>
#include <assert.h>
#include "vectors.h"

//-------------------------------------------------------------------------

template<class element>
vector<element>::vector(const int size)
{
    assert(size >= 0);
    Size = size;
    Elements = new element[size];
    assert(Elements != NULL);
}

//-------------------------------------------------------------------------

template<class element>
vector<element>::~vector(void)
{
    delete [] Elements;
}

//-------------------------------------------------------------------------

template<class element>
int vector<element>::size(void) const
{
    return Size;
}

//-------------------------------------------------------------------------

template<class element>
element& vector<element>::operator[](const int index) const
{
    assert(0 <= index && index < Size);
    return Elements[index];
}

/*************************************************************************/
```

## D.3 matrices.h

```
#ifndef MATRICES
#define MATRICES
/*************************************************************************/
/*                                                                       */
/*                 Skyline-Matrix Class Template Declaration             */
/*                 -----------------------------------------             */
/*                                                                       */
/* Written by: John Thornley, Computer Science Dept., Caltech.           */
/* Last modified: Tuesday 30th November 1993.                            */
/*                                                                       */
/* Generic skyline-matrix[0..size-1][0..size-1] class:                   */
/*                                                                       */
/* - Size and shape are arguments of the constructor.                    */
/* - Index values must lie in the range 0..size-1 and within the shape.  */
/* - Copy and assignment operations are prohibited.                      */
/*                                                                       */
/*************************************************************************/

template<class element>
class matrix_row;

template<class element>
class matrix {
public:
    matrix(const int size, const int row[], const int column[]);
    ~matrix(void);

    int size(void) const;
    int* row(void) const;
    int* column(void) const;
    matrix_row<element> operator[](const int first_index) const;
private:
    int Size;
    int* Row;
    int* Column;
    element** Lower; // Lower triangle rows (excluding diagonal).
    element** Upper; // Upper triangle columns (including diagonal).

    matrix(void);
    matrix(const matrix<element> &copy_from);
    matrix<element>& operator=(const matrix<element> &assign_from);

    friend class matrix_row<element>;
};

template<class element>
class matrix_row {
public:
    element& operator[](const int second_index) const;
private:
    const matrix<element>* Parent;
    int First_Index;
    matrix_row(const matrix<element> *const parent, const int first_index);
```

```
    matrix_row(const matrix_row<element> &copy_from);

    matrix_row(void);
    matrix_row<element>& operator=(const matrix_row<element> &assign_from);

    friend class matrix<element>;
};

/*****************************************************************************/
#endif // MATRICES
```

## D.4 matrices.C

```
/*************************************************************************/
/*                                                                       */
/*                Skyline-Matrix Class Template Definition               */
/*                ----------------------------------------               */
/*                                                                       */
/* Written by: John Thornley, Computer Science Dept., Caltech.           */
/* Last modified: Tuesday 30th November 1993.                            */
/*                                                                       */
/*************************************************************************/

#include <iostream.h>
#include <stdio.h>
#include <assert.h>
#include "matrices.h"

//-------------------------------------------------------------------------

template<class element>
matrix<element>::matrix(const int size, const int row[], const int column[])
{
    assert(size >= 0);
    Size = size;
    Row = new int[size];
    Column = new int[size];
    Lower = new element*[size];
    Upper = new element*[size];
    assert(Row != NULL && Column != NULL && Lower != NULL && Upper != NULL);
    for (int i = 0; i < size; i++) {
        assert(0 <= row[i] && row[i] <= i);
        Row[i] = row[i];
        assert(0 <= column[i] && column[i] <= i);
        Column[i] = column[i];
        Lower[i] = new element[i-row[i]];
        Upper[i] = new element[i-column[i]+1];
        assert(Lower[i] != NULL && Upper[i] != NULL);
    }
}

//-------------------------------------------------------------------------

template<class element>
matrix<element>::~matrix(void)
{
    delete [] Row;
    delete [] Column;
    for (int i = 0; i < Size; i++) {
        delete [] Lower[i];
        delete [] Upper[i];
    }
    delete [] Lower;
    delete [] Upper;
}
```

```
//---------------------------------------------------------------------------

template<class element>
int matrix<element>::size(void) const
{
    return Size;
}

//---------------------------------------------------------------------------

template<class element>
int* matrix<element>::row(void) const
{
    return Row;
}

//---------------------------------------------------------------------------

template<class element>
int* matrix<element>::column(void) const
{
    return Column;
}

//---------------------------------------------------------------------------

template<class element>
matrix_row<element> matrix<element>::operator[](const int first_index) const
{
    assert(0 <= first_index && first_index < Size);
    return matrix_row<element>(this, first_index);
}

//---------------------------------------------------------------------------

template<class element>
matrix_row<element>::matrix_row(const matrix<element> *const parent,
                                const int first_index)
{
    Parent = parent;
    First_Index = first_index;
}

//---------------------------------------------------------------------------

template<class element>
element& matrix_row<element>::operator[](const int second_index) const
{
    assert(0 <= First_Index && First_Index < Parent->Size);
    assert(0 <= second_index && second_index < Parent->Size);
    if (First_Index > second_index) {
        assert(second_index >= Parent->Row[First_Index]);
        return Parent->Lower[First_Index]
                            [second_index - Parent->Row[First_Index]];
    } else {
```

```
        assert(First_Index >= Parent->Column[second_index]);
        return Parent->Upper[second_index]
                            [First_Index - Parent->Column[second_index]];
    }
}

//---------------------------------------------------------------------------

template<class element>
matrix_row<element>::matrix_row(const matrix_row<element> &copy_from)
{
    Parent = copy_from.Parent;
    First_Index = copy_from.First_Index;
}

/***************************************************************************/
```

## D.5 solve.h

```
#ifndef SOLVE
#define SOLVE
/*************************************************************************/
/*                                                                       */
/*                      Solve Function Declaration                       */
/*                      --------------------------                       */
/*                                                                       */
/* Written by: John Thornley, Computer Science Dept., Caltech.           */
/* Last modified: Tuesday 30th November 1993.                            */
/*                                                                       */
/* DESCRIPTION:                                                          */
/*                                                                       */
/* Solve the following system of linear equations:                       */
/*                                                                       */
/*                                 Ax = b                                */
/*                                                                       */
/* without pivoting, where A is an n-by-n skyline matrix.  A skyline matrix  */
/* has nonzero values in row i in columns row[i] through i, and nonzero val- */
/* ues in column j in rows column[j] through j, where row and column are     */
/* vectors of size n.                                                    */
/*                                                                       */
/* PROBLEM FROM:                                                         */
/*                                                                       */
/* "A Comparative Study of Parallel Programming Languages: The Salishan      */
/* Problems", edited by John T. Feo.  Special Topics in Supercomputing,      */
/* Volume 6.  North-Holland, Amsterdam, 1992.                            */
/*                                                                       */
/*************************************************************************/

#include "vectors.h"
#include "matrices.h"

void solve(const matrix<sync float> &A, const vector<sync float> &b,
           vector<sync float> &x);
// Input Condition:
//     A.size() >= 0 and
//     A.size() = b.size() and A.size() = x.size() and
//     for all i in 0 .. n - 1 :
//         (0 <= A.row()[i] <= i and 0 <= A.column()[i] <= i) and
//     for all i in 0 .. n - 1, j in A.row()[i] .. i : A[i][j] != 0 and
//     for all j in 0 .. n - 1, i in A.column()[j] .. j : A[i][j] != 0 and
//     A is non-singular.
// Output Condition:
//     A*x = b.

/*************************************************************************/
#endif // SOLVE
```

## D.6 solve.C

```
/*************************************************************************/
/*                                                                       */
/*                        Solve Function Definition                      */
/*                        -------------------------                      */
/*                                                                       */
/* Written by: John Thornley, Computer Science Dept., Caltech.           */
/* Last modified: Tuesday 30th November 1993.                            */
/*                                                                       */
/*************************************************************************/

#include "vectors.h"
#include "matrices.h"
#include "solve.h"

//-------------------------------------------------------------------------

int max(const int x, const int y)
{
    if (x > y) return x; else return y;
}

//-------------------------------------------------------------------------

void LU_factorize(const matrix<sync float> &A, matrix<sync float> &LU)
{
    parfor (int i = 0; i < LU.size(); i++) par {
        parfor (int j = LU.row()[i]; j < i; j++) {
            float innerproduct;

            innerproduct = 0.0;
            for (int k = max(LU.row()[i], LU.column()[j]); k < j; k++)
                innerproduct += LU[i][k]*LU[k][j];
            LU[i][j] = (A[i][j] - innerproduct)/LU[j][j];
        }
        parfor (int J = LU.column()[i]; J <= i; J++) {
            float innerproduct;

            innerproduct = 0.0;
            for (int k = max(LU.row()[J], LU.column()[i]); k < J; k++)
                innerproduct += LU[J][k]*LU[k][i];
            LU[J][i] = A[J][i] - innerproduct;
        }
    }
}

//-------------------------------------------------------------------------

void forward_substitute(const matrix<sync float> &L,
                        const vector<sync float> &b, vector<sync float> &y)
{
    parfor (int i = 0; i < L.size(); i++) {
        float innerproduct;
```

```
        innerproduct = 0.0;
        for (int j = L.row()[i]; j < i; j++)
            innerproduct += L[i][j]*y[j];
        y[i] = b[i] - innerproduct;
    }
}

//---------------------------------------------------------------------------

void backward_substitute(const matrix<sync float> &U,
                         const vector<sync float> &y, vector<sync float> &x)
{
    parfor (int i = U.size() - 1; i >= 0; i--) {
        float innerproduct;

        innerproduct = 0.0;
        for (int j = i + 1; j < U.size(); j++)
            if (i >= U.column()[j])
                innerproduct += U[i][j]*x[j];
        x[i] = (y[i] - innerproduct)/U[i][i];
    }
}

//---------------------------------------------------------------------------

void solve(const matrix<sync float> &A, const vector<sync float> &b,
           vector<sync float> &x)
{
    matrix<sync float> LU(A.size(), A.row(), A.column());
    vector<sync float> y(A.size());

    par {
        LU_factorize(A, LU);           // Solve A = LU for L and U.
        forward_substitute(LU, b, y);  // Solve Ly = b for y.
        backward_substitute(LU, y, x); // Solve Ux = y for x.
    }
}

/***************************************************************************/
```